Type-safe Runtime Code Generation with LLVM

Trevor L. McDonell[1,2] Manuel M. T. Chakravarty[2] Vinod Grover[3] Ryan R. Newton[1]

[1]Indiana University Bloomington {mcdonelt,rrnewton}@indiana.edu

[2]University of New South Wales {tmcdonell,chak}@cse.unsw.edu.au

[3]NVIDIA Corporation vgrover@nvidia.com

Abstract

Embedded languages are often compiled at application runtime; thus, *embedded compile-time errors* become *application runtime errors*. We argue that advanced type system features, such as GADTs and type families, play a crucial role in minimising such runtime errors. Specifically, a rigorous type discipline reduces runtime errors due to bugs in both embedded language applications and the implementation of the embedded language compiler itself.

In this paper, we focus on the safety guarantees achieved by type preserving compilation. We discuss the compilation pipeline of *Accelerate*, a high-performance array language targeting both multicore CPUs and GPUs, where we are able to preserve types from the source language down to a low-level register language in SSA form. Specifically, we demonstrate the practicability of our approach by creating a new type-safe interface to the industrial-strength LLVM compiler infrastructure, which we used to build two new Accelerate backends that show competitive runtimes on a set of benchmarks across both CPUs and GPUs.

1. Introduction

Compiling a source language via a typed intermediate language has compelling advantages over a conventional untyped compiler. Carrying types can enable optimisations [38, 50], and it also helps ensure compiler correctness. An optimising compiler for a high-level language makes many passes over a single source program, performing sophisticated and error-prone transformations — many compiler bugs can be caught by type checking the intermediate language after each transformation.

Several practical compilers today, including the Glasgow Haskell Compiler (GHC), carry types through most or all of their compilation pipeline. These types, however, are represented at the *value level* inside the compiler. That is, the compiler's abstract syntax datatypes would include data constructors to distinguish, say, ints from floats, such as:

```
data Ty  = TyInt | TyFloat | ···
data Exp = Let (Var,Ty,Exp) Exp | ···
```

This approach has several drawbacks: (1) as the program progresses through the various compiler transformations, the value-level types must be carefully manipulated to remain in sync with the terms they annotate and (2) errors are only detected when the type checker or verifier is run over the intermediate representation,[1] which amounts to *testing* the compiler for a given user program, not *verifying* that the compiler preserves well-typedness in the intermediate language on all possible inputs. Thus, bugs can lurk undetected [54].

In Haskell, GADTs can be used to add a *type level* index to an expression syntax tree — yielding `Exp ty` rather than just `Exp`. In fact, this is the *canonical* example of how and why to use a GADT in Haskell. Fully deploying the technique, however, requires a full type-level representation of binding structure, which is rarely done in any sizable compiler. *Accelerate* [13, 37] is the only example of a released compiler with users that employs this technique, of which we are aware.

Unfortunately, even GADT-based compilers, including Accelerate (before this work), often stumble at the finish line: *code generation*. That is, type-preservation is lost at the point where C, assembly, or bytecode is emitted —typically by appending strings together— and this has empirically been the point where the most bugs creep in.[2] Of course, heavy-weight verification and proof-carrying-code mechanisms can address this [29, 33], but they require a vastly larger amount of effort. Moreover, these techniques have not yet been scaled to high performance and parallelizing compilers, which are the target of our work.

A small number of popular compilers, such as clang/LLVM and gcc, are debugged by the sheer force of many users. On the other hand, for young languages (e.g., Swift, Julia, Idris, Rust) this is not feasible, and embedded or domain specific languages provide an especially extreme case of many new compilers with small user bases. In fact, most parallelization-oriented DSLs developed over the last several years are not robust and complete. (Indeed, this is even true of OpenCL implementations! [15]) Thus we argue that new compilers for embedded languages deserve more effort to establish their correctness, *even if* for performance an unverified (but widely-used) backend such as LLVM, C, or CUDA must be part of the trusted code base.

Fixing the last mile for embedded languages. GADT techniques are most readily applicable to embedded languages, because type-level information is acquired "for free" from the host language type checker. Yet, there remains the problem of type-safe code generation. To that end, in this paper, we present a new, type-safe interface to LLVM code generation, for use by any Haskell-based compiler. We use this interface to build a family of backends for the Accelerate compiler [13, 37], resulting in complete type-

[1] In the case of GHC, this is only done while running GHC's regression test suite. CoreLint (GHC's internal type checker) is switched off during production use due to performance considerations.

[2] Example github issues: 37, 45, 50, 57, 66, 79, 91, 93, 114, 124, and 168, not to mention the many bugs that we ourselves found before release.

preservation from source to code generation, and targeting either CPUs or GPUs (compiling through LLVM to x86-64 and PTX, respectively).

We argue the result is a sweet spot that offers high confidence in compiler correctness relative to the amount of engineering effort required. Our method doesn't require any tools beyond the type system of Haskell itself.[3] In this method we trust the widely used LLVM compiler, but we verify type preservation for 100% of the (much less widely used) Accelerate compiler. While these are two very different methods of assurance, we find this to be a good combination.

We make the following contributions:

- We introduce a type-safe interface to the Haskell bindings for LLVM code generation. This can be used by future code generators in Haskell to increase confidence in generated code.
- We describe a series of static type-preserving transformations and optimisations—including mapping type representations, fusion, and code skeleton generation—from a parallel source program through to type-preserving generation of LLVM IR.
- We present a new backend for the Accelerate embedded language, based on those transformations, that targets both multicore CPUs and GPUs. To our knowledge, this is the first practical backend for an embedded language that targets an industrial-strength code generator, such as LLVM, while preserving all type information from the source to its low-level target language.
- We evaluate the performance of the new backend to validate that we have not sacrificed performance for safety in this effort.

This paper expands our existing work on the embedded language Accelerate [13, 37]. The source code is available from `http://github.com/AccelerateHS/accelerate-llvm`.

2. Background: Accelerate

Accelerate is a parallel language consisting of a carefully selected set of operations on multidimensional arrays that can be compiled efficiently to bulk-parallel SIMD hardware. Accelerate is *embedded* in Haskell, meaning that we write Accelerate programs using (slightly stylised) Haskell syntax. Accelerate code embedded into Haskell is not compiled to parallel SIMD code by the Haskell compiler; instead, the Accelerate library includes a *runtime compiler* that generates parallel SIMD code at application runtime. Accelerate is stratified into collective *array computations*, represented by terms of the type `Acc a`, where `a` is the type of the value produced by evaluating the expression, and *scalar expressions*, wrapped in the type constructor `Exp`. Collective operations comprise many scalar operations that are executed in parallel, but scalar operations *cannot* initiate new collective operations. This stratification statically excludes *nested, irregular parallelism*, which helps ensure efficient execution on constrained hardware such as GPUs, as discussed in our previous work [13].

Overall, the collective operations in Accelerate are based on the scan-vector model [14, 48], and consist of multidimensional variants of familiar Haskell list operations such as `map` and `fold`, as well as array-specific operations such as index permutations. For example, to compute a vector dot product, we write:

```
dotp :: Num a ⇒ Vector a → Vector a
     → Acc (Scalar a)
dotp xs ys =
    let xs' = use xs
        ys' = use ys
    in fold (+) 0 ( zipWith (*) xs' ys' )
```

The function `dotp` consumes two one-dimensional arrays (`Vector`) of values, and produces a single (`Scalar`) result as output. As the return type is wrapped in the type `Acc`, we see that it is an embedded Accelerate computation — it will be evaluated in the *object* language of dynamically generated parallel code, rather than the *meta* language, which is vanilla Haskell.

The arguments to `dotp` are plain Haskell arrays. To make these arguments available to Accelerate computations, they must be embedded with the `use` function, which is overloaded so that it can accept tuples of arrays:

```
use :: Arrays a ⇒ a → Acc a
```

The above Haskell code is more concise than an explicitly GPU-accelerated or SIMD vectorized[4] low-level dot product, but it is crucial to employ aggressive *fusion* [17, 37] to eliminate intermediate data structures, which would otherwise ruin performance.

As a second example, the code in Figure 1 computes a single round of the MD5 cryptographic hash function for a 512-bit input. The input is a two-dimensional array of words, with one word per column and padding (to 64 bytes) to keep the array regular. The hash computation is applied in data-parallel to all words in the input. Note that `md5Round` uses the host language Haskell as a meta language to *programatically* generate a single large expression in the object-language Accelerate. The meta program even includes such gratuitousness as list indexing (`!!`), operations that would ruin performance if we would try to compile them to parallel SIMD code. However, by taking advantage of Accelerate's runtime compilation, the Haskell meta program can generate an efficient Accelerate computation: embedding the various constant values directly into the generated program, and using `foldl`—from Haskell's standard prelude—to completely unroll the loops of the object code implementing the MD5 hash function.

This code clearly demonstrates the value of runtime code generation for embedded languages. Nevertheless, it also opens the door for runtime failures that users would expect were weeded out at compile time. In fact, many embedded array languages in less strongly typed languages [1, 9, 25], explicitly *allow* the runtime compiler to be a partial function that may fail to generate parallel code at runtime—sometimes falling back to *interpreted* sequential execution. We would instead prefer static assurances at meta-program (Haskell) compile time. By typing embedded programs using the type system of the host language and guaranteeing that we preserve that type information through the *entire* compilation process down to low-level register code, we minimise the likelihood of such failures without compromising performance.

3. Background: LLVM

Compiler backends and code generators are complex beasts, especially if they include advanced code optimisations and target multiple architectures. As a wide range of code optimisation and code generation techniques are largely independent of the specifics of the implemented source language and the corresponding compiler frontend, it is very attractive to reuse and share complex backend code. LLVM is probably the most popular and widely used set of libraries and tools to facilitate such backend reuse. It is applied well beyond its original target—the family of languages supported by GCC [31]—and now includes Java and .NET [22], Python [2],

[3] Further, the Haskell type system is all you need. In previous work [52] we recast Accelerate in Agda, and found few additional benefits from moving to a dependently typed language.

[4] We use the term "vectorized" to refer to a program that utilises the SSE/AVX instruction set extensions for x86 processors.

```
md5Round :: Acc (Array DIM2 Word32)
         → Exp Int
         → Exp (Word32,Word32,Word32,Word32)
md5Round dictionary word =
  lift $ foldl step (a0,b0,c0,d0) [0..64]
  where
    step (a,b,c,d) i
      | i < 16    = shfl ((b .&. c) .|.
                          ((complement b) .&. d))
      | i < 32    = shfl ((b .&. d) .|.
                          (c .&. (complement d)))
      | i < 48    = shfl (b `xor` c `xor` d)
      | i < 64    = shfl (c `xor`
                          (b .|. (complement d)))
      | otherwise = (a+a0,b+b0,c+c0,d+d0)
      where
        shfl f = (d, b + ((a + f + k i + get i)
                          `rotateL` r i), b, c)

    a0, b0, c0, d0 :: Exp Word32
    a0 = ···                  -- constants
```

```
get :: Int → Exp Word32
get i
  | i < 16    = get32le i
  | i < 32    = get32le ((5*i + 1) `rem` 16)
  | i < 48    = get32le ((3*i + 5) `rem` 16)
  | otherwise = get32le ((7*i)     `rem` 16)

get32le :: Int → Exp Word32
get32le i = dict A.! index2 (constant i) word

k :: Int → Exp Word32
k i = constant (ks !! i)
  where
    ks :: [Word32]
    ks = [ ··· ]          -- constants

r :: Int → Exp Int
r i = constant (rs !! i)
  where
    rs :: [Int]
    rs = [ ··· ]          -- constants
```

Figure 1. MD5 cryptographic hash computation in Accelerate

Ruby [3] and Haskell [51]. It is also being used for special purpose languages, such as NVIDIA's CUDA compiler for GPGPU computing [39].

LLVM has been designed from the outset as a compiler framework. Compared to generating architecture-specific code or generating a portable low-level language, such as C, LLVM has the following advantages:

- *Architecture support:* LLVM has cross-compilation support for a range of architectures, including x86[_64], PowerPC, and ARM. Moreover, it has support for high-throughput instruction sets, such as AVX-512, Intel's Xeon Phi, and PTX [26, 39, 42].
- *Optimisation passes:* LLVM implements a large number of compiler optimisations, including those that require machine specific knowledge. Individual optimisations can be chosen and ordered at compile time, and new optimisation passes can be dynamically loaded.
- *Online compilation:* LLVM offers several online compilation options, including an interpreter and JIT compiler. This is ideally suited to Accelerate programs, which are generated and optimised at runtime.
- *Language representation:* Operations in LLVM are represented in static single-assignment (SSA) [18] form, where every variable is assigned to once and never updated. Given the well-known correspondence between SSA and λ-calculus [5, 28], LLVM's intermediate language is a convenient target for the purely functional Accelerate language. Moreover, it allows us to avoid certain representation problems that we encountered in our original CUDA backend [13, 37].

3.1 The problem with Accelerate backends

Although Accelerate was designed with support for multiple architectures in mind—such as CPUs, GPUs, and even FPGAs—so far, only two complete backends have materialised: the interpreter, which only serves as a reference implementation for the semantics of the language, and the CUDA backend targeting individual GPUs [13, 37]. However, this is not from a lack of interest. Indeed, there exists no fewer than five incomplete or abandoned Accelerate backends, targeting Repa,[5] OpenCL,[6,7] and C.[8] In a sense this is not surprising: writing high-performance compilers is difficult and time consuming.

Unsurprisingly, it is those parts of the compiler that fail to preserve static type information that are the hardest to get right. The code generator of the original CUDA backend has been a large source of errors—we conjecture that many of these could have been avoided if the translation had preserved static types.

3.2 The LLVM Intermediate Representation

LLVM IR is a strongly-typed, low-level language in static single-assignment (SSA) format. It consists of sequences of register instructions such as add, subtract, and branch, operating over an infinite set of temporaries of the form `%0`, `%1`.... For example, the following defines a function `map` that loops over an input array `xs`, adding one to each element and storing the result into the array `ys`.

```
define void @map(i64 %ix.start, i64 %ix.end, float* %ys, float* %xs) {
entry:
  %0 = icmp slt i64 %ix.start, %ix.end
  br i1 %0, label %for1.top, label %for1.exit

for1.top:
  %1 = phi i64 [ %6, %for1.top ], [ %ix.start, %entry ]
  %2 = getelementptr float* %xs, i64 %1
  %3 = load float* %2
  %4 = fadd float 1.000000e+00, %3
  %5 = getelementptr float* %ys, i64 %1
  store float %4, float* %5
  %6 = add i64 %1, 1
  %7 = icmp slt i64 %6, %ix.end
  br i1 %7, label %for1.top, label %for1.exit

for1.exit:
  ret void
}
```

[5] `https://github.com/blambo/accelerate-repa`

[6] `https://github.com/HIPERFIT/accelerate-opencl/`

[7] `https://github.com/AccelerateHS/accelerate-backend-kit/tree/master/icc-opencl`

[8] `https://github.com/AccelerateHS/accelerate-c` and the project from the previous footnote.

Instructions are annotated with the type of their operands—`float` for single-precision floating point numbers, `i64` for (signed or unsigned) 64-bit integers, `i1` for `Bool`, and so forth. Applying an instruction such as `fadd` to operands of incorrect type is an error and is checked throughout the LLVM compilation pipeline.

LLVM IR can take three isomorphic forms: the above human-readable textual representation, a dense binary *bitcode* for serialisation, and an in-memory data structure, which all LLVM transformations passes use internally. Our goal is to statically guarantee that we only generate well-typed, in-memory LLVM IR, while simultaneously assuring a range of higher-level properties as discussed next.

4. Type Preservation

Accelerate is embedded in Haskell. More precisely, as illustrated by the examples of Section 2, Accelerate programs are comprised of Haskell expressions of type `Acc a` (representing embedded data-parallel array computations) and `Exp e` (representing embedded scalar computations). Hence, the Haskell compiler assigns types to Accelerate programs by way of type checking the Haskell code representing an Accelerate program. For example, consider incrementing each element of a vector of `Float`s:

```
inc :: Acc (Vector Float) → Acc (Vector Float)
inc = map (+1)
```

Here, the `map` is not Haskell's standard function on lists, but rather Accelerate's cousin operating on arrays of arbitrary rank:

```
map :: (Shape ix, Elt a, Elt b)
    ⇒ (Exp a → Exp b)
    → Acc (Array ix a)
    → Acc (Array ix b)
```

For a given *array shape* or *index domain* `ix` and array elements of type `a` and `b`, it takes a *scalar* Accelerate function of type `Exp a → Exp b` and an embedded array computation `Acc (Array ix a)` producing an `ix`-dimensional array of `a`s to produce a new embedded array computation that yields an array of the same dimensionality, but with elements of type `b`. In our example `inc`, `map` is used on a `Vector`, which is simply a one-dimensional (rank-1) array:

```
type Vector e = Array DIM1 e
```

The operator `(+)` is our old friend of the `Num` type class by way of an instance with head:

```
instance (Elt t, IsNum t) ⇒ Num (Exp t)
```

In other words, numeric operators may be used with scalar Accelerate expressions provided the values produced by those expressions are valid array element types `Elt t` and members of a type class `IsNum` discussed in the next subsection. (Accelerate needs to restrict elements types with `Elt` as only a limited number of types and their computations can be represented efficiently as data-parallel GPU operations.)

The first phase of Accelerate's type-preserving compilation pipeline reifies Accelerate programs —i.e., expressions of type `Acc a` and `Exp e`— as data structures in Haskell without losing any type information. It is well known that this can be achieved by the use of *Generalised Algebraic Data Structures (GADTs)* [6, 11, 41]. We take this a step further by also typing the embedded program's binding structure — a technique that originated from the realm of programming with dependent types [4].

4.1 Typed AST

To appreciate the representation of our running example `map (+1)`, it is important to remember that the section `(+1)` is a Haskell shorthand for the lambda abstraction `\x → x + 1`, which in de Bruijn form is $\lambda\ \mathbf{0}$ `+ 1`, where $\mathbf{0}$ represents the innermost lambda bound variable. This leads us to the following definition of `inc`, after it has been reified:

```
inc :: Acc (Vector Float) → Acc (Vector Float)
inc arr =
 Acc $
  Map
   (Lam (Body                  — first argument to map
    PrimAdd (⟨IsNum Float dictionary⟩)
    `PrimApp`
    Tuple (NilTup
          `SnocTup` (Var ZeroIdx) — de Bruijn idx 0
          `SnocTup` (Exp (Const 1)))))
    arr                        — second argument to map
```

The data constructor `Map` represents the `map` function, `Lam` introduces a de Bruijn binder, and `Body` marks a function body. The argument of `Var` is a typed de Bruijn index represented as a GADT:

```
data Idx env t where
  ZeroIdx ::               Idx (env, t) t
  SuccIdx :: Idx env t → Idx (env, s) t
```

Such an index projects a type `t` out of the type level environment `env` ensuring that bound variables are used at the correct type. This representation provides strong guarantees about the correct use of bound variables under program transformations (as we will discuss in the following section). It eliminates a common source of errors. The conversion from higher-order abstract syntax (HOAS) to de Bruijn form goes hand in hand with sharing recovery as detailed in [37].

`PrimAdd` represents uncurried addition, which, by way of `PrimApp`, is being applied to a pair of its two arguments. We represent tuples as snoc lists. This simplifies the rest of the code generator as we do not have to deal with n-tuples, but only with nested pairs. These are semantically isomorphic as all Accelerate functions and compound data types are strict.

Following the canonical implementation of type classes in Haskell [24, 40], we pass explicit dictionaries to overloaded functions, such as addition represented by `PrimAdd`. We discuss this next.

4.2 Reified type dictionaries

When GHC desugars Haskell programs after type checking, it turns type class constraints into explicit method dictionary parameters and the application of overloaded functions into the application of the function to a dictionary determined by the selected type instance. GHC desugars into a variant of System F, whereas Accelerate has to stay within Haskell, and hence, uses GADTs to represent type class dictionaries, while preserving full type information.

We achieve this by class constraints, such as `IsNum`, which we previously encountered in the `Num` instance for `Exp a`:

```
class (Num a, IsScalar a) ⇒ IsNum a where
  numType :: NumType a
```

Here `NumType` is a GADT that reifies the dictionary for the `Num` type class. This enables latter stages of the Accelerate compilation pipeline to vary code generation on the basis of the concrete types at which overloaded functions have been used. As reified dictionaries are data types, pattern matching suffices. Moreover, as we do not merely use plain data types, but GADTs —i.e., type indexed types— we can statically ensure that the code generator emits low-level operations at the appropriate low-level types.

An alternative design would be to implement the code generator by way of overloaded functions that are, directly or indirectly,

members of type classes, such as `IsNum`. While this would ensure type safety, it would compromise modularity. The Accelerate frontend presents an open interface that allows anybody to write backends *without the need to alter the Accelerate frontend.* In our experience, writing a variety of backends, different backends need support functions of differing types. To make that type safe, we would need to extend frontend classes whenever a backend needs a new such function. (An unacceptable alternative are *type erasing* query functions.) In contrast, typed dictionary reification enables us to preserve types and achieve modularity at the same time.

It turns out to be convenient to reify the class hierarchy as well. Hence,

```
data NumType a where
  IntegralNumType :: IntegralType a → NumType a
  FloatingNumType :: FloatingType a → NumType a
```

This distinguishes between members that belong to Haskell's `Integral` and `Floating` types, which enumerate primitive types; for example,

```
data IntegralType a where
  TypeInt  :: IntegralDict Int  → IntegralType Int
  TypeInt8 :: IntegralDict Int8 → IntegralType Int8
```

We have similar groupings such as Bounded or non-numeric types (such as `Char` and `Bool`) to establish a hierarchy of types with reified dictionaries that allows us to precisely specify which types are valid at each operation.

For the benefit of Accelerate's interpreter, which executes Accelerate programs by directly evaluating the AST, the constructors of the leaf types include all Haskell dictionaries (class instances) needed to execute overloaded Accelerate functions; for example,

```
data IntegralDict a where
  IntegralDict :: ( Num a, Eq a, ··· )
               ⇒ IntegralDict a
```

This ensures the same level of type safety for the interpreter as for the code generators.

4.3 Surface versus representation types

GPUs are very efficient at processing arrays of elementary types, such as integral and floating point types. They are significantly less efficient at chasing pointers and similar. In fact, the situation for CPUs is similar and becomes even more so with the increasing computational power of SIMD instruction sets, such as AVX. Consequently, Accelerate's array data type is not parametric. The type class `Elt` determines (a) the set of admissible *surface types* of array elements and (b) it prescribes a mapping of surface types to *representation types* used during code generation. For example, n-tuples are mapped to nested pairs and the custom data types for array shapes and indices (`Z`, `(:.)`, and so on) are similarly mapped to elementary types. In fact, the `Elt` class is *open*, and can also be extended to add new user-defined data types, as long as they can be mapped to suitable representation types.[9]

The set of low-level types that our code generators (and ultimately, LLVM) directly support is necessarily fixed. The type safe and extensible mapping from surface to representation types allows us more flexibility in the source language. The type class `Elt` implements the mapping by way of associated type synonyms and type families [12, 47], defined along the following lines:

```
type family EltRepr :: *

type instance EltRepr Int   = Int
type instance EltRepr Float = Float
type instance EltRepr (a,b) =
        ProdRepr ( EltRepr a, EltRepr b )
```

It builds on the type family `ProdRepr` that defines our canonical tuple format, representing products as heterogeneous snoc lists using `()` and `(,)` as type-level nil and cons respectively. Similarly to the `Elt` class, the `IsProduct` class encodes the conversion between the surface and representation type of products as nested pairs.

```
type instance ProdRepr (a,b)   = (((), a), b)
type instance ProdRepr (a,b,c) = ((((), a), b), c)
```

This encoding of products as type-level lists enables us to define type-safe tuple projection. Critically, while `Elt` is extensible, `EltRepr` consists only of scalar primitives, such as `Int` and `Float`, as well as units `()` and pairs `(,)`.

Note that `ProdRepr` prescribes the encoding of tuples into nested pairs at the top level only, whereas `EltRepr` performs the mapping all the way down to scalar types. This separation allows us to define a type-safe tuple projection in `ProdRepr`, but maintain a relationship between the two (non-injective) type encodings.

Much like the type class `IsNum`, which we discussed in the previous subsection, `Elt` includes a method

```
eltType :: a {− dummy −} → TupleType (EltRepr a)
```

that reifies the type of `a` as a GADT. This is for the same reasons as previously discussed for `IsNum`.

Moreover, we use very similar machinery to map array types of the surface language to array representation types that are effectively nested pairs of unboxed arrays of integral and floating-point types — implementing an unzipped ("struct-of-array") representation. The type mapping from surface to representation types used for element types and that for arrays of those element types are isomorphic. Moreover, we preserve the surface types in all program representations down to code generation. This leads to some correctness guarantees asserted by the Haskell compiler, while type checking the code generators. Firstly, we cannot confuse values that, despite having the same representation type, originate from different surface types. Secondly, low-level code that reads and writes elements in arrays that, by way of the unzipped ("struct-of-array") representation, are logically part of a single surface array does so in a consistent manner. Specifically, access to the various component arrays is always in lock step, compound element values are read and written whole, and similarly function arguments and results are maintained as prescribed by the type mapping.

5. Type Safe Optimisations

Historically, code optimisation is often problematic when asserting correctness properties of compilers [33]. However, experience with compilers using typed intermediate languages, and especially GHC, has demonstrated that code optimisation by transformations, as a series of localised, correctness preserving equational rewrites, facilitates a corresponding rewriting of types.

In our previous work [13], through a set of benchmarks, we identified the two most pressing performance limitations of Accelerate at the time: operator fusion and data sharing. This is not surprising as these are well known problems affecting functional array languages and deeply embedded languages, respectively. Hence, to achieve the safety guarantees of type preservation as well as efficient code, we need to go beyond previous work and realise type-preserving sharing recovery, common subexpression elimination, and a type-preserving fusion framework that produces code that is efficient on massively parallel SIMD hardware.

[9] For example, we define new `Elt` instances for the objects —spheres, planes, and light sources— used in the ray trace application (§8.3).

In previous work, we discussed our approach to type-safe sharing observation [37], but only outlined the computational aspects of our approach to array fusion without discussing type preservation. In the following, we describe how our approach to common subexpression elimination and our fusion system preserves types by adopting a transformational approach and reifying and tracking type equality.

5.1 Manipulating embedded programs

To apply transformations to well-typed terms while maintaining the properties of the program encoded in its type, we need to transform typed terms in a type- and binding-structure-preserving manner — we need to take care to manipulate types, type representations, and de Bruijn-style typed environments appropriately.

5.1.1 Propositional type equality

Consider the task of determining whether two subexpressions are equal, so that the duplicate computation can be eliminated:

```
λx → let a = x + 1
         b = x + 1
     in a + b
```

How should we implement `Exp s == Exp t`? If we don't care what `s` and `t` are, we can define standard heterogeneous equality as:

```
heq :: OpenExp env aenv s
    → OpenExp env aenv t
    → Bool
```

This signature requires the environment types of free scalar and array variables (called `env` and `aenv`, respectively) to have the same types, so that we can test equality of typed de Bruijn indices.

However, we often *do* care about the specific types of terms. Consider the case of moving under a let-binding, defined for scalar terms as:

```
data OpenExp env aenv t where
  Let :: (Elt bnd, Elt body)
      ⇒ OpenExp env         aenv bnd
      → OpenExp (env, bnd) aenv body
      → OpenExp env         aenv body
```

Here, the result type of the bound term —`bnd`— is existentially quantified, but to test equality of the body expression, we need to know something about this type in order to ensure that the scalar environments are compatible, namely `s`∼`bnd`∼`t`. In order to achieve this, our equality test must, in the positive case, deliver *evidence* that our types are equal:[10]

```
match :: OpenExp env aenv s
      → OpenExp env aenv t
      → Maybe (s :∼: t)      -- Just Refl on match
```

We compute the runtime witness justifying the equality of existentially quantified types by inspecting the reified dictionaries attached to our terms (§4.2). Now with this evidence-producing heterogeneous equality test, we can compare two terms and gain type-level knowledge when they witness the same value-level types. These witnesses allow us to test for equality *homogeneously*, and ensure that positive results from singleton tests give the bonus of unifying types for subsequent tests.

We use typed equality in the implementation of common subexpression elimination, constant propagation, and for other simplifying rewrites. We also use it to provide type witnesses during code generation.

[10] Using propositional equality from `Data.Type.Equality`.

5.1.2 Simultaneous substitution

To implement fusion, we need to be able to perform variable renaming (i.e., shifting of de Bruijn indices) and substitution by way of a type-preserving, value-level substitution algorithm. We closely follow McBride's method [36], which views both these operations as instances of a *single* traversal, pushing functions from variables to "stuff" through terms, for a suitable notion of "stuff". Moreover, we push a *type-preserving* but *environment-changing* operation `v` structurally through terms:

```
v :: ∀ t. Idx env t → stuff env' t
```

Where the operations differ is in the treatment of variables: renaming maps variables to variables, while substitution maps variables to terms. We lift this to an operation which traverses terms, lifting when pushing under bindings and rebuilding terms after applying `v` to the variables.

```
rebuild :: Syntactic stuff      -- variables and terms
        ⇒ (∀ t'. Idx env t' → stuff env' aenv t')
        → OpenExp env  aenv t
        → OpenExp env' aenv t
```

Overall, the crucial functionality of simultaneous substitution is to propagate a class of operations on variables closed under shifting. By choosing an appropriate function `v`, we define operations such as weakening, inlining, and function composition on terms; e.g.,

```
dot :: OpenExp (env, b) aenv c
    → OpenExp (env, a) aenv b
    → OpenExp (env, a) aenv c
dot f g = Let g (rebuild v f)
  where v :: Idx (env, b) c
          → OpenExp ((env, a), b) aenv c
        v = ···

compose :: OpenFun env aenv (b → c)
        → OpenFun env aenv (a → b)
        → OpenFun env aenv (a → c)
compose (Lam (Body f)) (Lam (Body g))
    = Lam (Body (f `dot` g))
```

5.2 Array Fusion

Most collective operations in Accelerate are array-to-array transformations. Our fusion algorithm proceeds in two phases: (1) *producer/producer*, a bottom-up contraction of the AST fuses sequences of producer operations into a single producer; and (2) *producer/consumer*, a top-down transformation that annotates the AST as to which nodes should be computed to manifest data, and which should be *delayed*, or embedded, into the operation which consumes them, so that their values are generated online without use of an intermediate array. The second phase is completed later in the compilation pipeline, when the code for the consumer operation is embedded directly into the skeleton template, so we do not need to consider this aspect further here. See [37] for background into the approach. The current presentation expands upon that work and presents those aspects of the algorithm that are relevant for type preservation.

5.2.1 Producer/Producer fusion

The basic idea behind our representation of fusable (producer) arrays in Accelerate is well known: represent an array by its size and a function mapping array indices to their corresponding values. This method has been used successfully to optimise purely functional array programs in Repa [27], but the idea is well known [19, 23]. We use the following typed representation of fusible producer arrays:

```
data Cunctation aenv a where
  Done  :: Arrays arrs
        ⇒ Idx        aenv arrs
        → Cunctation aenv arrs

  Yield :: (Shape sh, Elt e)
        ⇒ PreExp     aenv sh
        → PreFun     aenv (sh → e)
        → Cunctation aenv (Array sh e)

  Step  :: (Shape sh, Shape sh', Elt a, Elt b)
        ⇒ PreExp     aenv sh'
        → PreFun     aenv (sh' → sh)
        → PreFun     aenv (a   → b)
        → Idx        aenv (Array sh  a)
        → Cunctation aenv (Array sh' b)
```

Here, `Done` injects a manifest term into the type, while `Yield` and `Step` capture scalar functions that are used to construct an element at each index. Note that our definition is non-recursive — `Done` and `Step` are not defined in terms of array computations `Acc`, but instead carry a de Bruijn index `Idx` into the array environment.This allows our representation to be embedded within producer terms in the second phase, with the guarantee that an embedded scalar computation will not invoke further parallel computations.

The bottom-up contraction of the AST proceeds by converting terms into this representation, and merging sequences of producers into a single one. Smart constructors for each producer manage the integration with predecessor terms. Scalar functions are composed using the simultaneous substitution method described above (§5.1.2). For example, the smart constructor `mapD`, operating on the delayed representation, implements the well known fusion rule to reduce `map f . map g` sequences into `map (f . g)` is

```
mapD :: PreFun env aenv (a → b)
     → Cunctation aenv (Array sh a)
     → Cunctation aenv (Array sh b)
mapD f (Step sh p g v) = Step sh p (f `compose` g) v
mapD f (Yield sh g)    = Yield sh  (f `compose` g)
```

5.2.2 Removing obstacles

Equational fusion techniques need to be careful to spot fusion opportunities in cases where language constructs other than function application intervene between the two fusible operations. In Accelerate's internal language, the main obstacle is let bindings, as in this example:

```
map f $ let xs = use (Array ···)
        in map g xs
```

In this case, we want to *float* the let binding out to expose the producer chain for producer/producer fusion. In general, we float all let bindings of manifest data out across producer chains.

As the bottom-up contraction of the AST encounters manifest array data, we collect those terms into the following structure:

```
data Extend aenv aenv' where
  BaseEnv :: Extend aenv aenv
  PushEnv :: Extend aenv aenv'
          → OpenAcc     aenv' a
          → Extend aenv (aenv', a)
```

At the value level, this encodes a heterogeneous snoc-list of lifted-out terms, while the type captures how an array environment increases once we (eventually) bring these terms back into scope. Moreover, it provides a type *witness* for how to weaken a term — another simultaneous substitution (§5.1.2)— from one environment to another, where these new bindings have come into scope but no old bindings have disappeared.

```
sink :: Syntactic f
     ⇒ Extend env env' → f env t → f env' t
sink env = weaken (v env)
  where v BaseEnv       = id
        v (PushEnv e _) = SuccIdx . v e
```

Referring to our initial example, as we lift the binding of `xs` out through the outer term, `Extend` captures how to bring `map g` into the same environment type as `map f`, so that we can apply the `mapD` fusion rule from the previous subsection.

During AST contraction, our smart constructor for let-bindings examines the bound term and proceeds as follows: (1) if it is manifest data, add it to the list of floated-out terms stored in the `Extend` structure; (2) if the binding can be eliminated, inline the scalar fragments of the delayed array representation directly into the body term; otherwise, (3) keep the let-binding in place, being careful to maintain the structure of nested bindings, which would otherwise increase the scope of bound variables.

Finally, we note that separating the representation of delayed producers from the auxiliary binding structure is important for efficiency, so that we only `sink` a term for (possible) fusion via our smart constructors once, rather than at every analysis site.

6. Type Safe Code Generation

6.1 Bringing static types to LLVM

LLVM's intermediate language (IR), in-memory, represents type information only as a *value-level* data structure, as is common in compilers. Instead, we want to track IR types as Haskell types in the LLVM Haskell binding, such that we can statically guarantee to only generate type correct LLVM programs — avoiding LLVM type errors at application runtime. To this end, we use GADTs to define the LLVM instruction set:

```
data Instruction a where
  Add :: NumType a
      → Operand a
      → Operand a
      → Instruction a
```

Here, an `Operand` is an argument to an instruction, and can either be local references (such as the temporaries `%1`, `%2` that we saw in Section 3.2), or constant values, and are defined in a similar manner using type-safe GADTs. Instructions in this representation carry reified dictionaries (§4.2) that can be inspected to reveal which concrete type the instruction was instantiated with.

From well-typed Accelerate terms, we generate a well-typed LLVM AST while preserving types. Only in the last step, when we hand the program over to the standard LLVM (C++) library, do we convert the LLVM types captured in the Haskell type system into LLVM value-level types. To do so, we build upon the existing `llvm-general` package,[11] which provides FFI bindings into the LLVM API to construct, manipulate, and compile the generated code. We reflect LLVM types as values using a *downcast* type class of the following form:

```
class Downcast typed untyped where
  downcast :: typed → untyped

instance Downcast (NumType a)     LLVM.Type
instance Downcast (Instruction a) LLVM.Instruction
```

[11] `http://hackage.haskell.org/package/llvm-general`

6.2 Representing complex types

Even when representing LLVM IR as GADTs and properly tracking types, individual LLVM instructions operate only on primitive types such as `Int` and `Float`. Hence, we need to establish a mapping between instructions on scalar values to the much more expressive set of types characterised by `Elt` — which also includes nested tuples and, moreover, is user extensible. As we discussed before, for the sake of modularity, we require a strict separation between the Accelerate frontend and the various backends. This is where the representation types, which form the codomain of the previously discussed type family `EltRepr`, come into play.

We define the LLVM IR representation of a surface type `a` by a type constructor `IR` that is parameterised by `a`. In its definition, we use the type family `EltRepr` to map the surface type `a` to its representation type `EltRepr a`, which in turn is the type combining the LLVM operands representing `a`.

```
data IR a where
  IR :: Operands (EltRepr a) → IR a
```

The constructor `Operands`, in turn, is a data type family wrapping well-typed LLVM operands. Due to `EltRepr`, `Operands` only needs to be defined for primitive types, units, and pairs (i.e, by the set of representation types), but in `IR` still supports the full range of scalar types characterised by `Elt`.

```
data family Operands :: *
data instance Operands () = OP_Unit
data instance Operands Int = OP_Int (Operand Int)
data instance Operands Int8 = OP_Int8 (Operand Int8)
  ...
data instance Operands (a,b)
    = OP_Pair (Operands a) (Operands b)
```

This mapping from surface to representation types effectively encodes aggregate structures as collections of multiple scalar values. As an example, a value of surface type `(Int, Float)` has representation type `(((), Int), Float)`, and a corresponding encoding into `IR` as

```
IR $ OP_Unit
       `OP_Pair` (OP_Int ⟨Operand Int⟩)
       `OP_Pair` (OP_Float ⟨Operand Float⟩)
```

The last piece in the puzzle is that we can convert terms from this encoding into individual LLVM operands serving as arguments to LLVM instructions by way of type-level evidence that the type `a` represents a scalar value. To do so, we use the reified dictionaries discussed in Section 4.2. We traverse them to determine the concrete type of a value; for example,

```
class IROP dict where
  op :: dict a → IR a → Operand a
  ir :: dict a → Operand a → IR a

instance IROP IntegralType where
  op (TypeInt  _) (IR (OP_Int  x)) = x
  op (TypeInt8 _) (IR (OP_Int8 x)) = x
```

This also explains why we require a data family for `Operands`. A type synonym family wouldn't have given us this one-to-one mapping.

6.3 Mapping Accelerate to LLVM IR

Finally, we have the pieces necessary to translate our well-typed Accelerate programs into well-typed LLVM programs. We continue our running example program `inc` from Section 4.1, showing how to translate each fragment of the lambda abstraction `\x → x + 1` into well-typed IR.

6.3.1 Primitive function applications

As discussed earlier, the addition operation is encoded with the constructor `PrimAdd`, representing uncurried addition, which by way of `PrimApp` is applied to its two arguments in pair form. To generate the corresponding LLVM instructions, overall we require:

```
llvmOfPrimFun
    :: PrimFun (a → b) → IR a → IR b
llvmOfPrimFun (PrimAdd t) = uncurry (add t)
  ...
```

Here, `uncurry` is overloaded to operate on the `IR` data structure. Primitive scalar operations carry a dictionary reifying the concrete type of their arguments —here `t` as an `IsNum Float` dictionary— which we can use as evidence to unpack `IR Float` into `Operand Float` using the method of the previous subsection. Armed with a pair of scalar operands, we can finally apply our well-typed LLVM instructions from Section 6.1.

```
add :: NumType a → IR a → IR a → IR a
add t (op → x) (op → y) = ir (Add t x y)
```

The next subsections discuss how to generate the arguments for the application, namely a fragment of type `IR (Float, Float)`.

6.3.2 Constants

Scalar constants are defined in Accelerate using the following GADT constructor:

```
Const :: Elt t ⇒ EltRepr t → OpenExp env aenv t
```

Here, `t` ranges over all types in `Elt`: it is not limited to elementary values. If `t` represents on aggregate type, the resulting `IR` will consist of multiple elementary constants.

We can examine the structure of the embedded constant value by reifying its type using `eltType` (§4.3). Pattern matching on the resulting GADT allows us to walk over the structure of the *representation* type of `t`, which consists of nested tuples formed from unit, pair, and scalar values.

```
constant :: TupleType a → a → Operands a
constant UnitTuple         ()
  = OP_Unit
constant (PairTuple tx ty) (x,y)
  = OP_Pair (constant tx x) (constant ty y)
constant (ScalarType dict) x
  = ...
```

When we encounter a scalar value we will be equipped with a reified dictionary `dict`, that can be inspected to uncover the concrete type of the value `x::a`, and inject it as a fragment of LLVM IR.

6.3.3 Tuples

Our primitive function application construct treats all operations as unary functions. Referring to our example `\x → x + 1`, we must create a pair consisting of the constant 1 and the innermost lambda bound variable `x`. Scalar tuples are defined in Accelerate using the following constructor:

```
Tuple :: (Elt t, IsProduct t)
      ⇒ Tuple (OpenExp env aenv) (ProdRepr t)
      → OpenExp env aenv t
```

The *type* `Tuple` represents a data structure reifying the structure of the `ProdRepr` type as a snoc-list constructed from `()` and `(,)`. Critically, since our definition of `EltRepr` captures its relationship to `ProdRepr`, the conversion becomes straightforward.

```
llvmOfTuple' :: TupleType t
             → Tuple (OpenExp env aenv) tup
             → Operands t
llvmOfTuple' UnitTuple        NilTup
  = OP_Unit
llvmOfTuple (PairTuple ta tb) (SnocTup a b)
  = OP_Pair ···
```

7. The Accelerate-LLVM Backend Framework

LLVM is a reusable framework, portable across diverse architectures, and in the same spirit, we introduce the Accelerate-LLVM backend *framework*: a set of reusable components that reduce the marginal cost of creating *future* Accelerate backends, increase maintainability by sharing as much code as possible, and enable *all* backends to share the type-safety benefits outlined in the previous section. We validated this approach by building two new Accelerate backends: (1) a vectorising multicore CPU backend, and (2) a new GPU backend.

7.1 Architecture-specific considerations

The Accelerate-LLVM framework facilitates the construction of backends targeting different hardware architectures by using LLVM IR as a common intermediate language. However, although LLVM is able to cross-compile to a variety of architectures, code portability still does not come for free. Our backend framework provides a set of reusable components for operations such as compilation, code generation, and execution, where architecture-specific behaviour is established through a set of classes parameterised by the architecture being targeted.

In order to produce efficient code on the CPU or GPU, we must generate LLVM that is specific to a target architecture, but, regardless of that architecture, we use the same intermediate language to represent the code. For example, consider the task of generating code for the function `map f xs`. Depending on the target, the behaviour of concurrent threads executing the program will be different: on a multicore CPU we can split the input into contiguous chunks and assign each thread a different piece, but on a GPU, threads must process the array cooperatively in order to maintain memory coalescing and avoid SIMD divergence. As with our existing CUDA backend [13], code generation is based around the idea of algorithmic skeletons [16]. A backend implementer encodes the behaviour of each collective operation by instantiating the following class, where `arch` identifies the backend by way of a specific target architecture:

```
class Skeleton arch where
  map :: (Shape sh, Elt a, Elt b)
      ⇒ arch
      → Gamma          aenv
      → IRFun1    arch aenv (a → b)
      → IRDelayed arch aenv (Array sh a)
      → CodeGen (IROpenAcc arch aenv (Array sh b))
```

To compile a collective operation such as `map`, the Accelerate-LLVM framework generates code for each of the parameters of the skeleton, such as the scalar function applied at each element,[12] and instantiates the skeleton using the template provided by the backend implementor. Overall, the Accelerate-LLVM framework is designed to expose only the architecture-specific parts of backend construction, while reusing common infrastructure such as the type-preserving translation of scalar expressions presented previously and minimising tedious operations such as AST traversals.

[12] Code generation for scalar expressions is also abstracted into an architecture-parameterised class, allowing individual backends to override the default behaviour of each expression primitive.

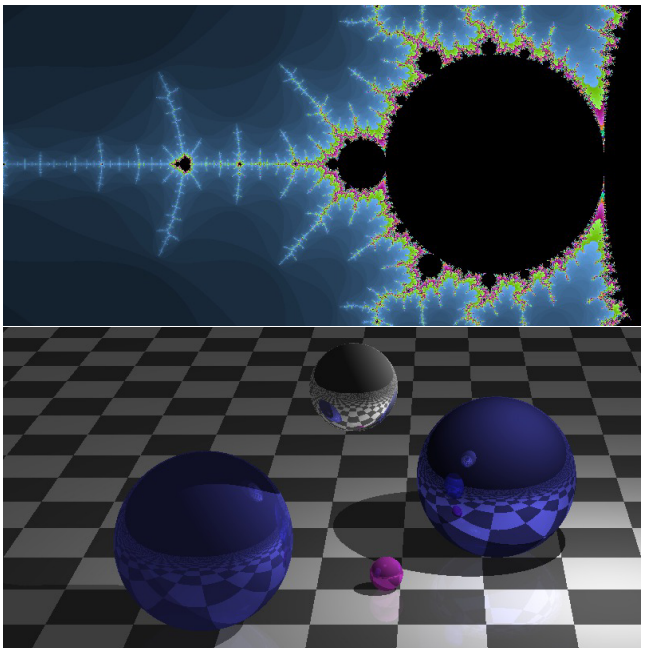


Figure 2. The Mandelbrot set (top) and a ray traced scene featuring multiple reflections (bottom). Both these workloads are unbalanced, as the time to compute each pixel varies.

7.2 Composable dynamic scheduling

Accelerate is aimed at high performance. Hence, we need to generate scalable code that can make effective use of increasing core counts. Static scheduling of regular array operations with many independent computations, such as `map f xs`, is easy: the number of elements in the input `xs` can be divided by the number of processors at runtime to yield the number of elements to be assigned to each core. While this works well, when each application of the function `f` completes within approximately the same amount of time, it results in load imbalance and poor performance when each processor performs differing and unpredictable amounts of work.

Figure 2 shows two example applications that exhibit unbalanced workloads. The first is a Mandelbrot set visualisation computed with the escape-time algorithm. In the output image, the pixels rendered black take longer to compute than all the others. The second image is the output from a real-time ray tracer, where those parts of the image showing many reflections take longer to compute than others. Although both of these examples are known in the folklore as being "embarrassingly parallel", as each pixel is computed independently of all others, they do not exhibit *regular* data parallelism due to the unbalanced workloads.

We address such unbalanced workloads by using dynamic scheduling based on *work stealing* [8], which has gained popularity for its good performance, ease of implementation, and theoretical bounds on space and time. In work-stealing schedulers, each worker maintains a private work pool, synchronising with other workers only when the local work is exhausted. Most work on dynamic scheduling has focused on scheduling of nested task parallelism; for example, the parallel function calls of recursive divide-and-conquer algorithm such as Quicksort. Since Accelerate is restricted to flat data-parallelism only, we need only support scheduling of do-all loops.

The Accelerate-LLVM framework includes a set of reusable scheduler components that a backend author can compose in the style of Foltzer et al. [21]. It includes a scheduler based on lazy

binary-splitting [53], which improves on the eager, static splitting approaches used by, for example, Intel's Thread Building Blocks [44] and Cilk++ [32], and does not require per-loop tuning parameters.

8. Benchmarks

The objective of this paper is code safety by way of compilation with type preservation. However, in the domain of high-performance array languages, code safety is not going to be appreciated if it comes at the expense of performance. Hence, we summarise the performance of the new Accelerate-LLVM CPU and GPU backends with a set of not previously published benchmark results. A summary of those results is in Table 1, where the runtimes for CPU-based programs report the best result attained regardless of number of cores used.

Benchmarks were conducted using a single Tesla K40c GPU (compute capability 3.5, 15 multiprocessors = 2880 cores at 750MHz, 11GB RAM) backed by two 12-core Xeon E5-2670 CPUs (64-bit, 2.3GHz, 32GB RAM, hyperthreading is enabled) running GNU/Linux (Ubuntu 14.04 LTS). We used GHC-7.8.3, LLVM-3.4.2, and NVCC-6.5.12. Haskell programs are compiled via LLVM using the recommend set of flags for Repa programs,[13] and run with RTS options to set thread affinity and match the allocation size to the processor cache size.[14] CPU results are generated using criterion[15] via linear regression. In order to exclude differences between the runtime systems of our two GPU backends, we focus on generated code performance and report GPU results as mean kernel execution time.[16]

8.1 Black-Scholes option pricing

The Black-Scholes algorithm solves a partial differential equation for modelling a stock option under certain assumptions. It is a balanced workload across all elements of the input; hence, it provides us with an estimate of the overhead incurred due to the dynamic work scheduling strategy in the Accelerate-LLVM CPU backend.

Comparing the CPU-based implementations, Accelerate enjoys a significant performance advantage over Repa. Both Repa and Accelerate use a non-parametric representation for arrays, so both implementations operate over three input and two output arrays of unboxed data. We speculate that the performance discrepancy is because GHC does not include aliasing information in the LLVM code it generates, resulting in fewer optimisations being applied.

Comparing GPU-based implementations, the LLVM-based code is slightly slower than that produced via CUDA. Internally, the CUDA compiler is based on LLVM [39], but additionally includes its own set of proprietary (closed source) optimisation passes, which we believe account for the extra performance from NVIDIA's compiler.

8.2 Mandelbrot fractal

The Mandelbrot set is generated by sampling values c in the complex plane, and determining whether under iteration of the complex quadratic polynomial $z_{n+1} = z_n^2 + c$ that $|z_n|$ remains bounded however large n gets. In the image shown in Figure 2, each pixel corresponds to a point c in the complex plane, and its colour depends on the number of iterations n before the relation diverges.

The Mandelbrot program has no array-valued inputs, but produces a single output array of 32-bit words encoded as RGBA data. As there are no array aliasing issues, the LLVM optimiser is able to optimise the GHC produced Repa code as well as the code produced by Accelerate, and performance of both is very similar. However, the Repa code does experience drops in performance at several points; we speculate that Repa's static scheduling strategy of the unbalanced workload happens to place an unusually large proportion of the work onto a small number of the cores in these cases. Since we use a dynamic work-stealing based scheduler, we were able to more evenly distribute the work in this case.

Comparing GPU implementations, we find that the code produced by Accelerate is actually *faster* than that produced by NVIDIA's compiler. Examining the generated PTX (assembly) code, we find that LLVM (1) removed a branch in the inner loop, replacing a short-circuit boolean-and with a logical-and; (2) produced six fused floating-point multiply-add instructions in the loop, versus three for NVCC; and (3) requires only 29 registers per thread, compared to 34 for NVCC — this results in the LLVM code being able to launch the maximum number of threads, whereas the NVCC code is limited to 79.9% thread occupancy.We speculate that this difference is due to NVCC being based on an older version of LLVM (based on the behaviour of the closed-source NVVM optimisation module, which accepts only up to LLVM-2.9 syntax).

8.3 Ray tracer

Ray tracing is a technique for generating an image by tracing the path of light through pixels in an image plane and simulating the effects of its encounters with virtual objects. The sample scene is shown in Figure 2. The technique is capable of producing images with a high degree of realism, but has a high computational cost compared to scanline rendering methods. Since the path of each individual ray varies depending on the number of objects it encounters, the amount of work performed at each pixel varies. We believe that NVIDIA's proprietary optimisation module gives it an edge in performance relative to the LLVM GPU backend.

8.4 MD5 hash

The MD5 message-digest algorithm [43] is a cryptographic hash function producing a 128-bit hash value that can be used for cryptographic and data integrity applications. Figure 1 shows how to compute the hash for 512-bits of input (16×32-bit words) in Accelerate. We compare our CPU backend to Hashcat, the "self-proclaimed worlds fastest CPU-based password recovery tool" (according to Wikipedia). We performed benchmarking using Hashcat's benchmark mode, but as Hashcat is closed source, we cannot verify that this is a fair comparison. Hence, this comparison should only be taken as indicative of our code generator being competitive, but without final judgement of how it ranks versus Hashcat. One source of the good performance of the code that our backend generates is the SIMD vectorisation performed via LLVM.

9. Related work

Repa [27, 34] is a Haskell library for parallel array programming on shared-memory SMP machines with very good performance. Repa also uses the delayed/manifest representation split on which our `Cunctation` type is based. Repa is not based on an embedded language, but on library functions compiled by GHC's code generator, which preserves types, but only as values. Hence, a separate CoreLint pass, only used during regression testing, is needed for type checking. We provide a quantitative comparison in Section 8.

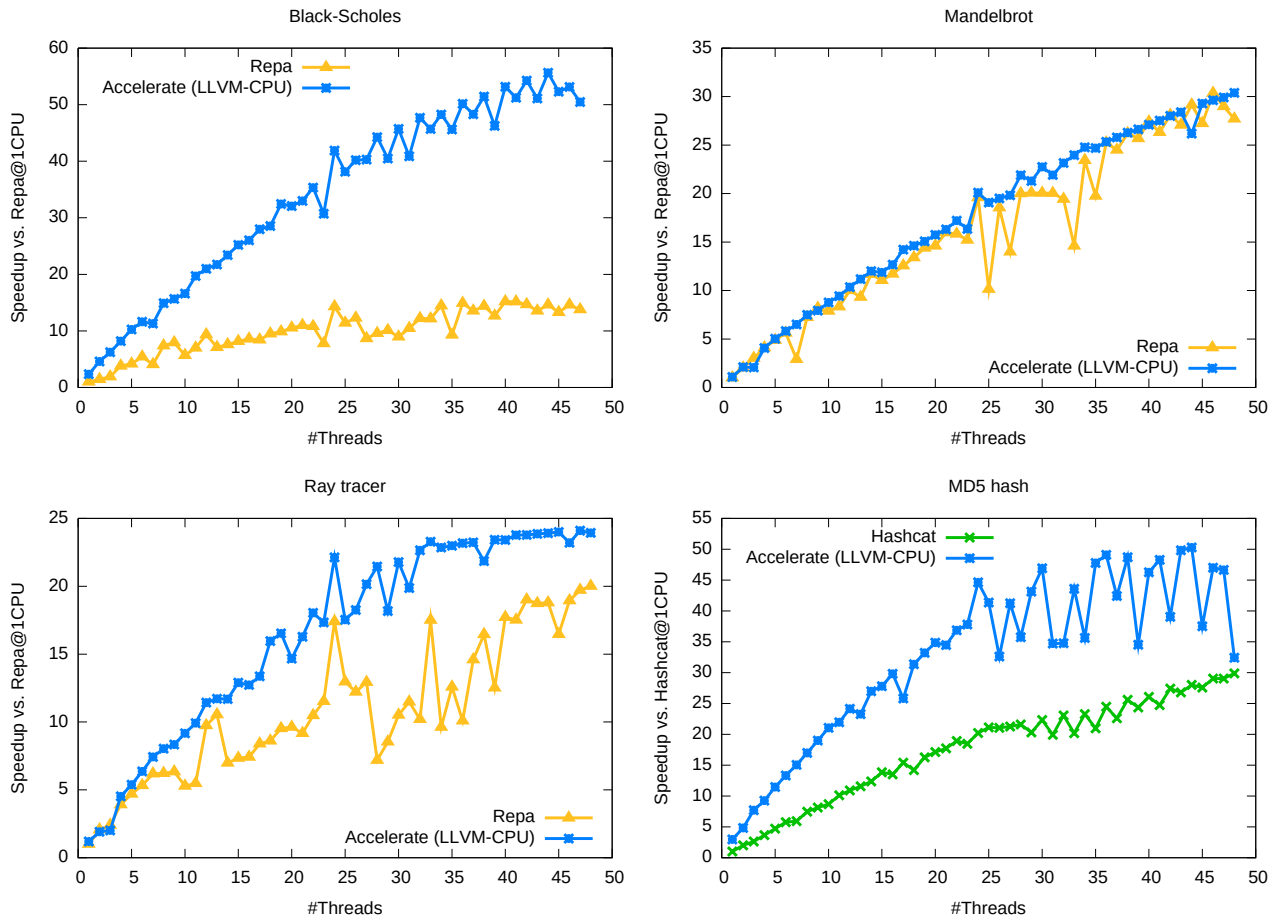
Vertigo [20], Nikola [35] and Obsidian [49] are EDSLs in Haskell that generate GPU code. None of these systems preserves source language types throughout the pipeline and none of them

[13] `-Odph -rtsopts -threaded -fno-liberate-case -funfolding-use-threshold1000 -funfolding-keeness-factor1000 -fllvm -optlo-O3`

[14] `-qa -A30M -N`n

[15] `http://hackage.haskell.org/package/criterion`

[16] The exact number of iterations is controlled via criterion to ensure (for the overall runtime) $R^2 \geq 0.99$.



Benchmark	Input size	Contender (ms)		Accelerate (CUDA) (ms)		Accelerate (LLVM-CPU) (ms)		Accelerate (LLVM-GPU) (ms)	
Black Scholes	20M	251.1	(Repa)	3.023	(1.20%)	68.53	(27.3%)	3.446	(1.37%)
Mandelbrot	2M	16.65	(Repa)	3.761	(22.6%)	16.63	(99.8%)	2.284	(13.7%)
Ray tracer	2M	29.57	(Repa)	5.065	(17.1%)	24.32	(82.3%)	10.80	(36.5%)
MD5 hash	14M	38.60	(Hashcat)	10.51	(27.2%)	22.93	(59.4%)	13.51	(35.0%)

Table 1. Benchmark summary

are able to generate CPU and GPU code, or currently supporting multiple backends. Moreover, Accelerate supports a significantly richer set of computations. Baracuda [30] is another Haskell EDSL that produces CUDA GPU kernels, though it is intended to be used offline, with the kernels being called directly from C++.

Delite/LMS [10, 45, 46] is a parallelisation framework for DSLs in Scala that uses library-based multi-pass staging to specify complex optimisations in a modular manner. Like Accelerate, Delite is a modular system that supports multiple code generators and targets CPU and GPU systems. It is not type preserving like Accelerate.

NDP2GPU [7] compiles NESL code down to CUDA. However, the source language is not embedded and no runtime code generation is performed.

There is ample previous work on type-preserving compilation (e.g., [38, 50]) and on full scale verification (e.g., [29, 33]). However, neither has so far been used from source to low-level code in a *runtime compiler* aimed at *high-performance*, nor has it been demonstrated for a practical embedded language.

References

[1] OpenACC. URL `http://www.openacc.org`.

[2] PyPy. URL `http://pypy.org`.

[3] Rubinius. URL `http://rubini.us`.

[4] T. Altenkirch and B. Reus. Monadic Presentation of Lambda Terms Using Generalised Inductive Types. In *CSL '99: Computer Science Logic*, pages 453–468, 1999.

[5] A. W. Appel. SSA is functional programming. *ACM SIGPLAN Notices*, 33(4):17–20, 1998.

[6] R. Atkey, S. Lindley, and J. Yallop. Unembedding domain-specific languages. In *Haskell Symposium*, pages 37–48, 2009.

[7] L. Bergstrom and J. Reppy. Nested data-parallelism on the GPU. In *ICFP: International Conference on Functional Programming*, 2012.

[8] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748, 1999.

[9] B. Catanzaro, M. Garland, and K. Keutzer. Copperhead: Compiling an embedded data parallel language. In *PPoPP: Principles and Practice*

of Parallel Programming, 2011.

[10] H. Chafi, Z. DeVito, A. Moors, T. Rompf, A. K. Sujeeth, P. Hanrahan, M. Odersky, and K. Olukotun. Language virtualization for heterogeneous parallel computing. In *OOPSLA: Object Oriented Programming Systems & Applications*, pages 835–847, 2010.

[11] M. M. T. Chakravarty. Converting a HOAS term GADT into a de Bruijn term GADT, 2009. URL `http://www.cse.unsw.edu.au/~chak/haskell/term-conv/`.

[12] M. M. T. Chakravarty, G. Keller, and S. Peyton Jones. Associated type synonyms. In *POPL: Principles of Programming Languages*, pages 241–253, 2005.

[13] M. M. T. Chakravarty, G. Keller, S. Lee, T. L. McDonell, and V. Grover. Accelerating Haskell array codes with multicore GPUs. In *DAMP: Declarative Aspects of Multicore Programming*, 2011.

[14] S. Chatterjee, G. E. Blelloch, and M. Zagha. Scan primitives for vector computers. In *Supercomputing*, pages 666–675, 1990.

[15] N. Chong, A. F. Donaldson, A. Lascu, and C. Lidbury. Many-Core Compiler Fuzzing. In *PLDI: Programming Language Design and Implementation*, 2015.

[16] M. I. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. The MIT Press, 1989.

[17] D. Coutts, R. Leshchinskiy, and D. Stewart. Stream fusion: from lists to streams to nothing at all. In *ICFP: International Conference on Functional Programming*, pages 315–326, 2007.

[18] R. Cryton, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, 1991.

[19] C. Elliott. Functional Images. In *The Fun of Programming*. Palgrave, 2003.

[20] C. Elliott. Programming graphics processors functionally. In *Haskell Symposium*, 2004.

[21] A. Foltzer, A. Kulkarni, R. Swords, S. Sasidharan, E. Jiang, and R. Newton. A meta-scheduler for the par-monad: composable scheduling for the heterogeneous cloud. In *ICFP: International Conference on Functional Programming*, 2012.

[22] N. Geoffray, G. Thomas, J. Lawall, G. Muller, and B. Folliot. VMKit: a substrate for managed runtime environments. In *Virtual Execution Environments*, 2010.

[23] L. J. Guibas and D. K. Wyatt. Compilation and Delayed Evaluation in APL. In *POPL: Principles of Programming Languages*, pages 1–8, 1978.

[24] C. V. Hall, K. Hammond, S. L. Peyton Jones, and P. L. Wadler. Type classes in Haskell. *ACM Transactions on Programming Languages and Systems*, 18(2):109–138, 1996.

[25] S. Herhut, R. L. Hudson, T. Shpeisman, and J. Sreeram. River trail: a path to parallelism in JavaScript. In *OOPSLA: Object Oriented Programming Systems & Applications*, pages 729–744, 2013.

[26] Intel Corporation. Intel SPMD Program Compiler. URL `http://ispc.github.io`.

[27] G. Keller, M. M. T. Chakravarty, R. Leshchinskiy, S. Peyton Jones, and B. Lippmeier. Regular, shape-polymorphic, parallel arrays in Haskell. In *ICFP: International Conference on Functional Programming*, pages 261–272, 2010.

[28] R. A. Kelsey. A correspondence between continuation passing style and static single assignment form. In *Workshop on Intermediate Representations*, pages 13–22, 1995.

[29] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal Verification of an OS Kernel. In *SOSP: Symposium on Operating Systems Principles*, pages 207–220. ACM, 2009.

[30] B. Larsen. Simple Optimizations for an Applicative Array Language for Graphics Processors. In *DAMP: Declarative Aspects of Multicore Programming*, 2011.

[31] C. Lattner and V. Adve. Architecture for a Next-Generation GCC. In *GCC Developers' Summit*, 2003.

[32] C. E. Leiserson. The Cilk++ concurrency platform. In *Design Automation Conference*, pages 522–527, 2009.

[33] X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107, 2009.

[34] B. Lippmeier, M. M. T. Chakravarty, G. Keller, and S. Peyton Jones. Guiding parallel array fusion with indexed types. In *Haskell Symposium*, 2012.

[35] G. Mainland and G. Morrisett. Nikola: Embedding Compiled GPU Functions in Haskell. In *Haskell Symposium*, 2010.

[36] C. McBride. Type-Preserving Renaming and Substitution. *Journal of Functional Programming*, 2006.

[37] T. L. McDonell, M. M. T. Chakravarty, G. Keller, and B. Lippmeier. Optimising Purely Functional GPU Programs. In *ICFP: International Conference on Functional Programming*, 2013.

[38] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to Typed Assembly Language. In *POPL: Principles of Programming Languages*, 1998.

[39] NVIDIA. CUDA LLVM Compiler. URL `https://developer.nvidia.com/cuda-llvm-compiler`.

[40] J. Peterson and M. Jones. Implementing type classes. In *PLDI: Programming Language Design and Implementation*, pages 227–236, 1993.

[41] S. Peyton Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple unification-based type inference for GADTs. In *ICFP: International Conference on Functional Programming*, pages 50–61, 2006.

[42] M. Pharr and W. R. Mark. ispc: A SPMD compiler for high-performance CPU programming. In *Innovative Parallel Computing*, pages 1–13, 2012.

[43] R. Rivest. The MD5 Message-Digest Algorithm, 1992.

[44] A. Robison, M. Voss, and A. Kukanov. Optimization via Reflection on Work Stealing in TBB. In *IPDPS: Parallel & Distributed Processing Symposium*, pages 1–8, 2008.

[45] T. Rompf, A. K. Sujeeth, N. Amin, K. J. Brown, V. Jovanovic, H. Lee, M. Jonnalagedda, K. Olukotun, and M. Odersky. Optimizing data structures in high-level programs: new directions for extensible compilers based on staging. In *POPL: Principles of Programming Languages*, 2013.

[46] T. Rompf, A. K. Sujeeth, K. J. Brown, H. Lee, H. Chafi, and K. Olukotun. Surgical precision JIT compilers. In *PLDI: Programming Language Design and Implementation*, pages 41–52, 2014.

[47] T. Schrijvers, S. Peyton Jones, M. M. T. Chakravarty, and M. Sulzmann. Type checking with open type functions. In *ICFP: International Conference on Functional Programming*, pages 51–62, 2008.

[48] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens. Scan primitives for GPU computing. In *Graphics Hardware*, pages 97–106, 2007.

[49] B. J. Svensson, M. Sheeran, and K. Claessen. Obsidian: A domain specific embedded language for parallel programming of graphics processors. In *IFL: Implementation and Application of Functional Languages*, 2008.

[50] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: a type-directed optimizing compiler for ML. In *PLDI: Programming Language Design and Implementation*, pages 181–192, 1996.

[51] D. A. Terei and M. M. T. Chakravarty. An LLVM backend for GHC. In *Haskell Symposium*, pages 109–120, 2010.

[52] P. Thiemann and M. M. T. Chakravarty. Agda Meets Accelerate. In *IFL: Implementation and Application of Functional Languages*, 2012.

[53] A. Tzannes, G. C. Caragea, R. Barua, and U. Vishkin. Lazy binary-splitting: a run-time adaptive work-stealing scheduler. In *PPoPP: Principles and Practice of Parallel Programming*, pages 179–190, 2010.

[54] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. In *PLDI: Programming Language Design and Implementation*, pages 283–294, 2011.